e-Notes by S.R.Mangalwade, GIT, Belgaum

Chapter 1. The UNIX Operating System

Introduction

This chapter introduces you to the UNIX operating system. We first look at what is an operating system and then proceed to discuss the different features of UNIX that have made it a popular operating system.

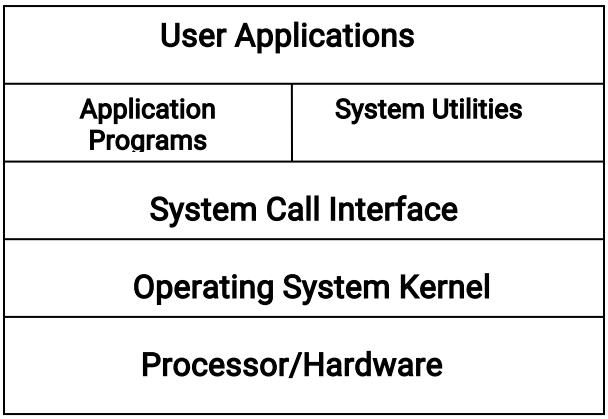
Objectives

- What is an operating system (OS)?
- Features of UNIX OS
- A Brief History of UNIX OS, POSIX and Single Unix Specification (SUS)

1. What is an operating system (OS)?

An operating system (OS) is a resource manager. It takes the form of a set of software routines that allow users and application programs to access system resources (e.g. the CPU, memory, disks, modems, printers, network cards etc.) in a **safe, efficient** and **abstract** way.

For example, an OS ensures **safe** access to a printer by allowing only one application program to send data directly to the printer at any one time. An OS encourages **efficient** use of the CPU by suspending programs that are waiting for I/O operations to complete to make way for programs that can use the CPU more productively. An OS also provides convenient **abstractions** (such as files rather than disk locations) which isolate application programmers and users from the details of the underlying hardware.



UNIX Operating system allows complex tasks to be performed with a few keystrokes. It doesn't tell or warn the user about the consequences of the command.

Kernighan and Pike (The UNIX Programming Environment) lamented long ago

that “as the UNIX system has spread, the fraction of its users who are skilled in its application has decreased.” However, the capabilities of UNIX are limited only by your imagination.

2. Features of UNIX OS

Several features of UNIX have made it popular. Some of them are:

Portable

UNIX can be installed on many hardware platforms. Its widespread use can be traced to the decision to develop it using the C language.

Multiuser

The UNIX design allows multiple users to concurrently share hardware and software

Multitasking

UNIX allows a user to run more than one program at a time. In fact more than one program can be running in the background while a user is working foreground.

Networking

While UNIX was developed to be an interactive, multiuser, multitasking system, networking is also incorporated into the heart of the operating system. Access to another system uses a standard communications protocol known as Transmission Control Protocol/Internet Protocol (TCP/IP).

Organized File System

UNIX has a very organized file and directory system that allows users to organize and maintain files.

Device Independence

UNIX treats input/output devices like ordinary files. The source or destination for file input and output is easily controlled through a UNIX design feature called redirection.

Utilities

UNIX provides a rich library of utilities that can be use to increase user productivity.

3. A Brief History of UNIX

In the late 1960s, researchers from General Electric, MIT and Bell Labs launched a joint project to develop an ambitious multi-user, multi-tasking OS for mainframe computers known as MULTICS (Multiplexed Information and Computing System). MULTICS failed, but it did inspire Ken Thompson, who was a researcher at Bell Labs, to have a go at writing a simpler operating system himself. He wrote a simpler version of MULTICS on a PDP7 in assembler and

called his attempt UNICS (Uniplexed Information and Computing System). Because memory and CPU power were at a premium in those days, UNICS (eventually shortened to UNIX) used short commands to minimize the space needed to store them and the time needed to decode them - hence the tradition of short UNIX commands we use today, e.g. ls, cp, rm, mv etc.

Ken Thompson then teamed up with Dennis Ritchie, the author of the first C compiler in 1973. They rewrote the UNIX kernel in C - this was a big step forwards in terms of the system's portability - and released the Fifth Edition of UNIX to universities in 1974. The Seventh Edition, released in 1978, marked a split in UNIX development into two main branches: SYSV (System 5) and BSD (Berkeley Software Distribution). BSD arose from the University of California at Berkeley where Ken Thompson spent a sabbatical year. Its development was continued by students at Berkeley and other research institutions. SYSV was developed by AT&T and other commercial companies. UNIX flavors based on SYSV have traditionally been more conservative, but better supported than BSD-based flavors.

Until recently, UNIX standards were nearly as numerous as its variants. In early days, AT&T published a document called System V Interface Definition (SVID). X/OPEN (now The Open Group), a consortium of vendors and users, had one too, in the X/Open Portability Guide (XPG). In the US, yet another set of standards, named Portable Operating System Interface for Computer Environments (POSIX), were developed at the behest of the Institution of Electrical and Electronics Engineers (IEEE).
In 1998, X/OPEN and IEEE undertook an ambitious program of unifying the two standards. In 2001, this joint initiative resulted in a single specification called the Single UNIX Specification, Version 3 (SUSV3), that is also known as IEEE 1003.1:2001 (POSIX.1). In 2002, the International Organization for Standardization (ISO) approved SUSV3 and IEEE 1003.1:2001.

Some of the commercial UNIX based on system V are:

- IBM's AIX
- Hewlett-Packard's HPUX
- SCO's Open Server Release 5
- Silicon Graphics' IRIS
- DEC's Digital UNIX
- Sun Microsystems' Solaris 2

Some of the commercial UNIX based on BSD are:

- SunOS 4.1.X (now Solaris)
- DEC's Ultris
- BSD/OS, 4.4BSD

Some Free UNIX are:

- Linux, written by Linus Torvalds at University of Helsinki in Finland.
- FreeBSD and NetBSD, a derivative of 4.4BSD

Conclusion

In this chapter we defined an operating system. We also looked at history of UNIX and features of UNIX that make it a popular operating system. We also discussed the convergence of different flavors of UNIX into Single Unix Specification (SUS) and Portable Operating System Interface for Computing Environments (POSIX).

Chapter 2. The UNIX Architecture and Command Usage

Introduction

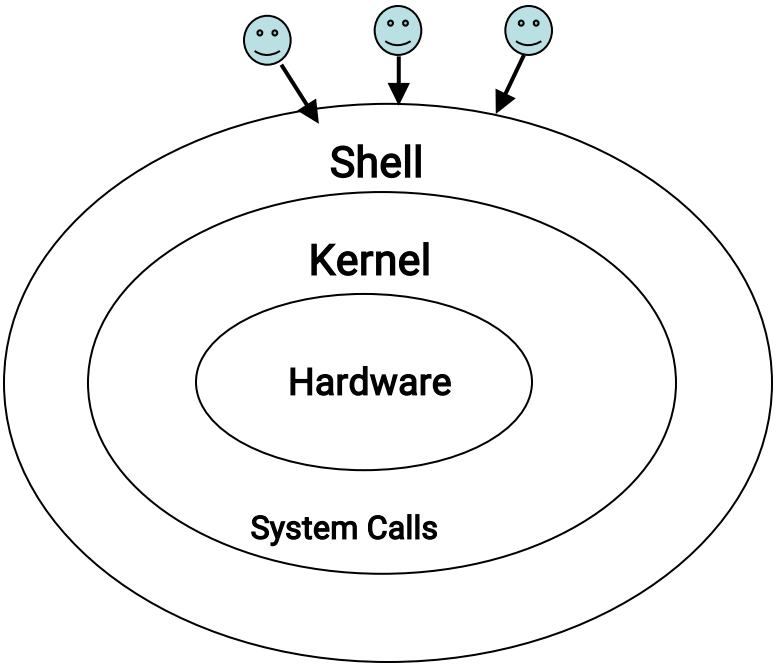
In order to understand the subsequent chapters, we first need to understand the architecture of UNIX and the concept of division of labor between two agencies viz., the shell and the kernel. This chapter introduces the architecture of UNIX. Next we discuss the rich collection of UNIX command set, with a specific discussion of command structure and usage of UNIX commands. We also look at the man command, used for obtaining online help on any UNIX command. Sometimes the keyboard sequences don't work, in which case, you need to know what to do to fix them. Final topic of this chapter is troubleshooting some terminal problems.

Objectives

- The UNIX Architecture
- Locating Commands
- Internal and External Commands
- Command Structure and usage
- Flexibility of Command Usage
- The man Pages, apropos and whatis
- Troubleshooting the terminal problems

1. The UNIX Architecture

Users



UNIX architecture comprises of two major components viz., the shell and the kernel. The kernel interacts with the machine's hardware and the shell with the user.

The kernel is the core of the operating system. It is a collection of routines written in C. It is loaded into memory when the system is booted and communicates directly with the hardware. User programs that need to access the hardware use the services of the kernel via use of system calls and the kernel performs the job on behalf of the user. Kernel is also responsible for managing system's memory, schedules processes, decides their priorities.

The shell performs the role of command interpreter. Even though there's only one kernel running on the system, there could be several shells in action, one for each user who's logged in. The shell is responsible for interpreting the meaning of metacharacters if any, found on the command line before dispatching the command to the kernel for execution.

The File and Process

A file is an array of bytes that stores information. It is also related to another file in the sense that both belong to a single hierarchical directory structure.

A process is the second abstraction UNIX provides. It can be treated as a time image of an executable file. Like files, processes also belong to a hierarchical structure. We will be discussing the processes in detain in a subsequent chapter.

2. Locating Files

All UNIX commands are single words like ls, cd, cat, etc. These names are in lowercase. These commands are essentially *files* containing programs, mainly written in C. Files are stored in directories, and so are the binaries associated with these commands. You can find the location of an executable program using type command:

```
$ type ls
ls is /bin/ls
```

This means that when you execute ls command, the shell locates this file in /bin directory and makes arrangements to execute it.

The Path

The sequence of directories that the shell searches to look for a command is specified in its own PATH variable. These directories are colon separated. When you issue a command, the shell searches this list in the sequence specified to locate and execute it.

3. Internal and External Commands

Some commands are implemented as part of the shell itself rather than separate executable files. Such commands that are built-in are called internal commands. If a command exists both as an internal command of the shell as well as an external one (in /bin or /usr/bin), the shell will accord top priority to its own internal command with the same name. Some built-in commands are echo, pwd, etc.

4. Command Structure

UNIX commands take the following general form:

```
verb [options] [arguments]
```

where verb is the command name that can take a set of optional options and one or more optional arguments.

Commands, options and arguments have to be separated by spaces or tabs to enable the shell to interpret them as words. A contiguous string of spaces and tabs together is called a whitespace. The shell compresses multiple occurrences of whitespace into a single whitespace.

Options

An option is preceded by a minus sign (-) to distinguish it from filenames.

Example: $ ls –l

There must not be any whitespaces between – and l. Options are also arguments, but given a special name because they are predetermined. Options can be normally compined with only one – sign. i.e., instead of using

```
$ ls –l –a –t
```

we can as well use,

```
$ ls –lat
```

Because UNIX was developed by people who had their own ideas as to what

options should look like, there will be variations in the options. Some commands use + as an option prefix instead of -.

Filename Arguments

Many UNIX commands use a filename as argument so that the command can take input from the file. If a command uses a filename as argument, it will usually be the last argument, after all options.

Example: cp file1 file2 file3 dest_dir
rm file1 file2 file3

The command with its options and argumens is known as the command line, which is considered as complete after *[Enter]* key is pressed, so that the entire line is fed to the shell as its input for interpretation and execution.

Exceptions

Some commands in UNIX like pwd do not take any options and arguments. Some commands like who may or may not be specified with arguments. The ls command can run without arguments (ls), with only options (ls –l), with only filenames (ls f1 f2), or using a combination of both (ls –l f1 f2). Some commands compulsorily take options (cut). Some commands like grep, sed can take an expression as an argument, or a set of instructions as argument.

5. Flexibility of Command Usage

UNIX provides flexibility in using the commands. The following discussion looks at how permissive the shell can be to the command usage.

Combining Commands

Instead of executing commands on separate lines, where each command is processed and executed before the next could be entered, UNIX allows you to specify more than one command in the single command line. Each command has to be separated from the other by a ; (semicolon).

```
wc sample.txt ; ls –l sample.txt
```

You can even group several commands together so that their combined output is redirected to a file.

```
(wc sample.txt ; ls –l sample.txt) > newfile
```

When a command line contains a semicolon, the shell understands that the command on each side of it needs to be processed separately. Here ; is known as a metacharacter.

Note: When a command overflows into the next line or needs to be split into multiple lines, just press enter, so that the secondary prompt (normally >) is displayed and you can enter the remaining part of the command on the next line.

Entering a Command before previous command has finished

You need not have to wait for the previous command to finish before you can enter the next command. Subsequent commands entered at the keyboard are

stored in a buffer (a temporary storage in memory) that is maintained by the kernel for all keyboard input. The next command will be passed on to the shell for interpretation after the previous command has completed its execution.

6. man: Browsing The Manual Pages Online

UNIX commands are rather cryptic. When you don't remember what options are supported by a command or what its syntax is, you can always view man (short for manual) pages to get online help. The man command displays online documentation of a specified command.

A pager is a program that displays one screenful information and pauses for the user to view the contents. The user can make use of internal commands of the pager to scroll up and scroll down the information. The two popular pagers are more and less. more is the Berkeley's pager, which is a superior alternative to original pg command. less is the standard pager used on Linux systems. less if modeled after a popular editor called vi and is more powerful than more as it provides vi-like navigational and search facilities. We can use pagers with commands like ls | more. The man command is configured to work with a pager.

7. Understanding The man Documentation

The man documentation is organized in eight (08) sections. Later enhancements have added subsections like 1C, 1M, 3N etc.) References to other sections are reflected as SEE ALSO section of a man page.

When you use man command, it starts searching the manuals starting from section 1. If it locates a keyword in one section, it won't continue the search, even if the keyword occurs in another section. However, we can provide the section number additionally as argument for man command.

For example, passwd appears in section 1 and section 4. If we want to get documentation of passwd in section 4, we use,

$ man 4 passwd OR $ man –s4 passwd (on Solaris)

Understanding a man Page

A typical man page for wc command is shown below:

User Commands wc(1)

NAME

wc – displays a count of lines, words and characters in a file

SYNOPSIS

wc [-c | -m | -C] [-lw] [file ...]

DESCRIPTION

The wc utility reads one or more input files and, by default, writes the number of newline characters, words and bytes contained in each input file to the standard output. The utility also writes a total count for all named files, if more than one input file is specified.

OPTIONS

The following options are supported:

-c Count bytes.

-m Count characters.

-C same as –m.

-l Count lines.

-w Count words delimited by white spaces or new line characters ...

OPERANDS

The following operand is supported:

file A path name of an input file. If no file operands are specified, the standard input will be used.

EXIT STATUS

See largefile(5) for the description of the behavior of wc when encountering files greater than or equal to 2 Gbyte (2 **31 bytes)

SEE ALSO

cksum(1), isspace(3C), iswalpha(3C), iswspace(3C), largefile(5), ...

A man page is divided into a number of compulsory and optional sections. Every command doesn't need all sections, but the first three (NAME, SYNOPSIS and DESCRIPTION) are generally seen in all man pages. NAME presents a one-line introduction of the command. SYNOPSIS shows the syntax used by the command and DESCRIPTION provides a detailed description.

The SYNOPSIS follows certain conventions and rules:

- If a command argument is enclosed in rectangular brackets, then it is optional; otherwise, the argument is required.
- The ellipsis (a set if three dots) implies that there can be more instances of the preceding word.
- The | means that only one of the options shows on either side of the pipe can be used.

All the options used by the command are listed in OPTIONS section. There is a separate section named EXIT STATUS which lists possible error conditions and their numeric representation.

Note: You can use man command to view its own documentation ($ man man). You can also set the pager to use with man ($ PAGER=less ; export PAGER). To understand which pager is being used by man, use $ echo $PAGER.

The following table shows the organization of man documentation.

Section	Subject (SVR4)	Subject (Linux)
1	User programs	User programs
2	Kernel's system calls	Kernel's system calls
3	Library functions	Library functions
4	Administrative file formats	Special files (in /dev)
5	Miscellaneous	Administrative file formats
6	Games	Games
7	Special files (in /dev)	Macro packages and conventions
8	Administration commands	Administration commands

8. Further Help with man –k, apropos and whatis

man –k: Searches a summary database and prints one-line description of the command.
Example:

```
$ man –k awk
awk     awk(1)          -pattern scanning and processing language
nawk    nawk(1)         -pattern scanning and processing language
```

apropos: lists the commands and files associated with a keyword.
Example:

```
$ apropos FTP
ftp       ftp(1)          -file transfer program
ftpd      in.ftpd(1m)     -file transfer protocol server
ftpusers        ftpusers(4) -file listing users to be disallowed
                                        ftp login privileges
```

whatis: lists one-liners for a command.
Example:

```
$ whatis cp
cp        cp(1)           -copy files
```

9. When Things Go Wrong

Terminals and keyboards have no uniform behavioral pattern. Terminal settings directly impact the keyboard operation. If you observe a different behavior from that expected, when you press certain keystrokes, it means that the terminal

settings are different. In such cases, you should know which keys to press to get the required behavior. The following table lists keyboard commands to try when things go wrong.

Keystroke or command	**Function**
[Ctrl-h]	Erases text
[Ctrl-c] or Delete	Interrupts a command
[Ctrl-d]	Terminates login session or a program that expects its input from keyboard
[Ctrl-s]	Stops scrolling of screen output and locks keyboard
[Ctrl-q]	Resumes scrolling of screen output and unlocks keyboard
[Ctrl-u]	Kills command line without executing it
[Ctrl-\]	Kills running program but creates a core file containing the memory image of the program
[Ctrl-z]	Suspends process and returns shell prompt; use **fg** to resume job
[Ctrl-j]	Alternative to *[Enter]*
[Ctrl-m]	Alternative to *[Enter]*
stty sane	Restores terminal to normal status

Conclusion

In this chapter, we looked at the architecture of UNIX and the division of labor between two agencies viz., the shell and the kernel. We also looked at the structure and usage of UNIX commands. The man documentation will be the most valuable source of documentation for UNIX commands. Also, when the keyboard sequences won't sometimes work as expected because of different terminal settings. We listed the possible remedial keyboard sequences when that happens.

Chapter 3. The File System

Introduction

In this chapter we will look at the file system of UNIX. We also look at types of files their significance. We then look at two ways of specifying a file viz., with absolute pathnames and relative pathnames. A discussion on commands used with directory files viz., cd, pwd, mkdir, rmdir and ls will be made. Finally we look at some of the important directories contained under UNIX file system.

Objectives

- Types of files
- UNIX Filenames
- Directories and Files
- Absolute and Relative Pathnames
- pwd – print working directory
- cd – change directory
- mkdir – make a directory
- rmdir – remove directory
- The PATH environmental variable
- ls – list directory contents
- The UNIX File System

1. Types of files

A simple description of the UNIX system is this:
"On a UNIX system, everything is a file; if something is not a file, it is a process."
A UNIX system makes no difference between a file and a directory, since a directory is just a file containing names of other files. Programs, services, texts, images, and so forth, are all files. Input and output devices, and generally all devices, are considered to be files, according to the syste2m.

Most files are just files, called *regular* files; they contain normal data, for example text files, executable files or programs, input for or output from a program and so on.
While it is reasonably safe to suppose that everything you encounter on a UNIX system is a file, there are some exceptions.
Directories: files that are lists of other files.
Special files or Device Files: All devices and peripherals are represented by files. To read or write a device, you have to perform these operations on its associated file. Most special files are in /dev.
Links: a system to make a file or directory visible in multiple parts of the system's file tree.
(Domain) sockets: a special file type, similar to TCP/IP sockets, providing inter−process networking protected by the file system's access control.
Named pipes: act more or less like sockets and form a way for processes to communicate with each other, without using network socket semantics.

Ordinary (Regular) File

This is the most common file type. An ordinary file can be either a text file or a binary file.

A text file contains only printable characters and you can view and edit them. All C and Java program sources, shell scripts are text files. Every line of a text file is terminated with the *newline* character.

A binary file, on the other hand, contains both printable and nonprintable characters that cover the entire ASCII range. The object code and executables that you produce by compiling C programs are binary files. Sound and video files are also binary files.

Directory File

A directory contains no data, but keeps details of the files and subdirectories that it contains. A directory file contains one entry for every file and subdirectory that it houses. Each entry has two components namely, the filename and a unique identification number of the file or directory (called the *inode number).*

When you create or remove a file, the kernel automatically updates its corresponding directory by adding or removing the entry (filename and inode number) associated with the file.

Device File

All the operations on the devices are performed by reading or writing the file representing the device. It is advantageous to treat devices as files as some of the commands used to access an ordinary file can be used with device files as well.

Device filenames are found in a single directory structure, /dev. A device file is not really a stream of characters. It is the attributes of the file that entirely govern the operation of the device. The kernel identifies a device from its attributes and uses them to operate the device.

2. Filenames in UNIX

On a UNIX system, a filename can consist of up to 255 characters. Files may or may not have extensions and can consist of practically any ASCII character except the / and the Null character. You are permitted to use control characters or other nonprintable characters in a filename. However, you should avoid using these characters while naming a file. It is recommended that only the following characters be used in filenames:

- Alphabets and numerals.
- The period (.), hyphen (-) and underscore (_).

UNIX imposes no restrictions on the extension. In all cases, it is the application that imposes that restriction. Eg. A C Compiler expects C program filenames to end with .c, Oracle requires SQL scripts to have .sql extension.

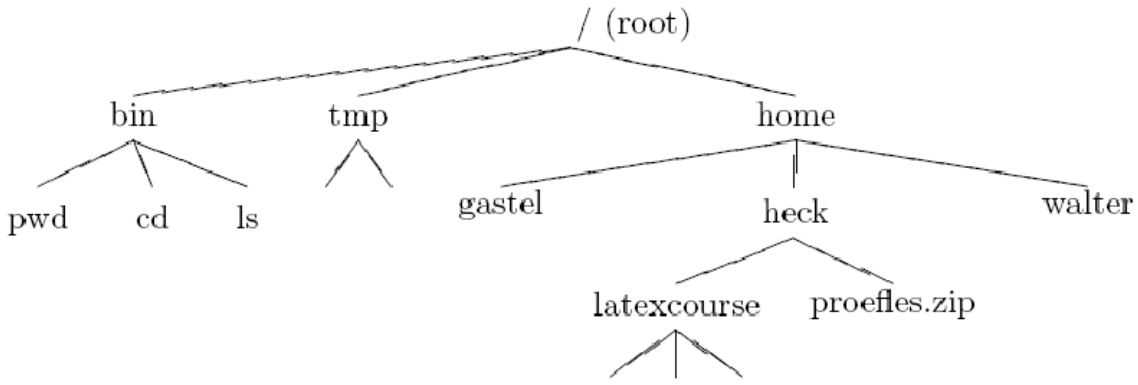
A file can have as many dots embedded in its name. A filename can also begin with or end with a dot.

UNIX is case sensitive; chap01, Chap01 and CHAP01 are three different

filenames that can coexist in the same directory.

3. Directories and Files

A file is a set of data that has a name. The information can be an ordinary text, a user-written computer program, results of a computation, a picture, and so on. The file name may consist of ordinary characters, digits and special tokens like the underscore, except the forward slash (/). It is permitted to use special tokens like the ampersand (&) or spaces in a filename.
Unix organizes files in a tree-like hierarchical structure, with the *root directory*, indicated
by a forward slash (/), at the top of the tree. See the Figure below, in which part of the hierarchy of files and directories on the computer is shown.



4. Absolute and relative paths

A path, which is the way you need to follow in the tree structure to reach a given file, can be described as starting from the trunk of the tree (the / or root directory). In that case, the path starts with a slash and is called an absolute path, since there can be no mistake: only one file on the system can comply.
Paths that don't start with a slash are always relative to the current directory. In relative paths we also use the . and .. indications for the current and the parent directory.

The HOME variable

When you log onto the system, UNIX automatically places you in a directory called the *home directory*. The shell variable HOME indicates the home directory of the user.

E.g.,

```
$ echo $HOME
  /home/kumar
```

What you see above is an absolute pathname, which is a sequence of directory names starting from root (/). The subsequent slashes are used to separate the directories.

5. pwd - print working directory

At any time you can determine where you are in the file system hierarchy with the ***pwd***, print working directory, command,

E.g.,:

```
$ pwd
/home/frank/src
```

6. cd - change directory

You can change to a new directory with the ***cd***, change directory, command. ***cd*** will accept both absolute and relative path names.

Syntax

cd [directory]

Examples

cd changes to user's home directory
cd / changes directory to the system's root
cd .. goes up one directory level
cd ../.. goes up two directory levels
cd /full/path/name/from/root changes directory to absolute path named (note the leading slash)
cd path/from/current/location changes directory to path relative to current location (no leading slash)

7. mkdir - make a directory

You extend your home hierarchy by making sub-directories underneath it. This is done with the ***mkdir***, make directory, command. Again, you specify either the full or relative path of the directory.

Examples

mkdir patch Creates a directory *patch* under current directory
mkdir patch dbs doc Creates three directories under current directory
mkdir pis pis/progs pis/data Creates a directory tree with *pis* as a directory under the current directory and *progs* and *data* as subdirectories under *pis*

Note the order of specifying arguments in example 3. The parent directory should be specified first, followed by the subdirectories to be created under it.

The system may refuse to create a directory due to the following reasons:
1. The directory already exists.
2. There may be an ordinary file by the same name in the current directory.
3. The permissions set for the current directory don't permit the creation of files and directories by the user.

8. rmdir - remove directory

A directory needs to be empty before you can remove it. If it's not, you need to remove the files first. Also, you can't remove a directory if it is your present working directory; you must first change out of that directory. You cannot remove a subdirectory unless you are placed in a directory which is hierarchically *above* the one you have chosen to remove.

E.g.

rmdir patch Directory must be empty

rmdir pis pis/progs pis/data Shows error as *pis* is not empty.

However rmdir silently deletes the lower level subdirectories *progs* and *data*.

9. The PATH environment variable

Environmental variables are used to provide information to the programs you use. We have already seen one such variable called HOME.

A command runs in UNIX by executing a disk file. When you specify a command like *date*, the system will locate the associated file from a list of directories specified in the PATH variable and then executes it. The PATH variable normally includes the current directory also.

Whenever you enter any UNIX command, you are actually specifying the name of an executable file located somewhere on the system. The system goes through the following steps in order to determine which program to execute:

1. Built in commands (such as cd and history) are executed within the shell.
2. If an absolute path name (such as /bin/ls) or a relative path name (such as ./myprog), the system executes the program from the specified directory.
3. Otherwise the PATH variable is used.

10. ls - list directory contents

The command to list your directories and files is ***ls***. With options it can provide information about the size, type of file, permissions, dates of file creation, change and access.

Syntax

ls [options] [argument]

Common Options

When no argument is used, the listing will be of the current directory. There are many very useful options for the ls command. A listing of many of them follows. When using the command, string the desired options together preceded by "**-**".

-a Lists all files, including those beginning with a dot (**.**).

-d Lists only names of directories, not the files in the directory

-F Indicates type of entry with a trailing symbol: executables with *, directories with / and symbolic links with @

-R Recursive list

-u Sorts filenames by last access time

-t Sorts filenames by last modification time
-i Displays inode number
-l Long listing: lists the mode, link information, owner, size, last modification (time). If the file is a symbolic link, an arrow (-->) precedes the pathname of the linked-to file.

The **mode field** is given by the **-l** option and consists of 10 characters. The first character is one of the following:

CHARACTER	IF ENTRY IS A
d	directory
-	plain file
b	block-type special file
c	character-type special file
l	symbolic link
s	socket

The next 9 characters are in 3 sets of 3 characters each. They indicate the **file access permissions**: the first 3 characters refer to the permissions for the **user**, the next three for the users in the Unix **group** assigned to the file, and the last 3 to the permissions for **other** users on the system.
Designations are as follows:

r read permission
w write permission
x execute permission
- no permission

Examples
1. To list the files in a directory:

```
$ ls
```

2. To list all files in a directory, including the hidden (dot) files:

```
$ ls -a
```

3. To get a long listing:

```
$ ls -al
total 24
drwxr-sr-x 5 workshop acs 512 Jun 7 11:12 .
drwxr-xr-x 6 root sys 512 May 29 09:59 ..
-rwxr-xr-x 1 workshop acs 532 May 20 15:31 .cshrc
-rw------- 1 workshop acs 525 May 20 21:29 .emacs
-rw------- 1 workshop acs 622 May 24 12:13 .history
-rwxr-xr-x 1 workshop acs 238 May 14 09:44 .login
-rw-r--r-- 1 workshop acs 273 May 22 23:53 .plan
-rwxr-xr-x 1 workshop acs 413 May 14 09:36 .profile
-rw------- 1 workshop acs 49 May 20 20:23 .rhosts
drwx------ 3 workshop acs 512 May 24 11:18 demofiles
drwx------ 2 workshop acs 512 May 21 10:48 frank
drwx------ 3 workshop acs 512 May 24 10:59 linda
```

11. The UNIX File System

The root directory has many subdirectories. The following table describes some of the subdirectories contained under root.

Directory	Content

/bin	Common programs, shared by the system, the system administrator and the users.
/dev	Contains references to all the CPU peripheral hardware, which are represented as files with special properties.
/etc	Most important system configuration files are in /etc, this directory contains data similar to those in the Control Panel in Windows
/home	Home directories of the common users.
/lib	Library files, includes files for all kinds of programs needed by the system and the users.
/sbin	Programs for use by the system and the system administrator.
/tmp	Temporary space for use by the system, cleaned upon reboot, so don't use this for saving any work!
/usr	Programs, libraries, documentation etc. for all user-related programs.
/var	Storage for all variable files and temporary files created by users, such as log files, the mail queue, the print spooler area, space for temporary storage of files downloaded from the Internet, or to keep an image of a CD before burning it.

Conclusion

In this chapter we looked at the UNIX file system and different types of files UNIX understands. We also discussed different commands that are specific to directory files viz., pwd, mkdir, cd, rmdir and ls. These commands have no relevance to ordinary or device files. We also saw filenaming conventions in UNIX. Difference between the absolute and relative pathnames was highlighted next. Finally we described some of the important subdirectories contained under root (/).

Chapter 6. The Shell

Introduction

In this chapter we will look at one of the major component of UNIX architecture – The Shell. Shell acts as both a command interpreter as well as a programming facility. We will look at the interpretive nature of the shell in this chapter.

Objectives

- The Shell and its interpretive cycle
- Pattern Matching – The wild-cards
- Escaping and Quoting
- Redirection – The three standard files
- Filters – Using both standard input and standard output
- /dev/null and /dev/tty – The two special files
- Pipes
- tee – Creating a tee
- Command Substitution
- Shell Variables

1. The shell and its interpretive cycle

The shell sits between you and the operating system, acting as a command interpreter. It reads your terminal input and translates the commands into actions taken by the system. The shell is analogous to ***command.com*** in DOS. When you log into the system you are given a default shell. When the shell starts up it reads its startup files and may set environment variables, command search paths, and command aliases, and executes any commands specified in these files. The original shell was the Bourne shell, ***sh***. Every Unix platform will either have the Bourne shell, or a Bourne compatible shell available.

Numerous other shells are available. Some of the more well known of these may be on your Unix system: the Korn shell, ***ksh***, by David Korn, C shell, ***csh***, by Bill Joy and the Bourne Again SHell, ***bash***, from the Free Software Foundations GNU project, both based on ***sh***, the T-C shell, ***tcsh***, and the extended C shell, ***cshe***, both based on ***csh***.

Even though the shell appears not to be doing anything meaningful when there is no activity at the terminal, it swings into action the moment you key in something.

The following activities are typically performed by the shell in its interpretive cycle:

- The shell issues the prompt and waits for you to enter a command.
- After a command is entered, the shell scans the command line for metacharacters and expands abbreviations (like the * in rm *) to recreate

a simplified command line.

- It then passes on the command line to the kernel for execution.
- The shell waits for the command to complete and normally can't do any work while the command is running.
- After the command execution is complete, the prompt reappears and the shell returns to its waiting role to start the next cycle. You are free to enter another command.

2. Pattern Matching – The Wild-Cards

A pattern is framed using ordinary characters and a metacharacter (like *) using well-defined rules. The pattern can then be used as an argument to the command, and the shell will expand it suitably before the command is executed.

The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards. The following table lists them:

Wild-Card	Matches
*	Any number of characters including none
?	A single character
[ijk]	A single character – either an i, j or k
[x-z]	A single character that is within the ASCII range of characters x and z
[!ijk]	A single character that is not an i,j or k (Not in C shell)
[!x-z]	A single character that is not within the ASCII range of the characters x and x (Not in C Shell)
{pat1,pat2…}	Pat1, pat2, etc. (Not in Bourne shell)

Examples:

To list all files that begin with *chap*, use

```
$ ls chap*
```

To list all files whose filenames are six character long and start with chap, use

```
$ ls chap??
```

Note: Both * and ? operate with some restrictions. for example, the * doesn't match all files beginning with a . (dot) ot the / of a pathname. If you wish to list all hidden filenames in your directory having at least three characters after the dot, the dot must be matched explicitly.

```
$ ls .???*
```

However, if the filename contains a dot anywhere but at the beginning, it need not be matched explicitly.

Similarly, these characters don't match the / in a pathname. So, you cannot use

```
$ cd /usr?local        to change to /usr/local.
```

The character class

You can frame more restrictive patterns with the character class. The character class comprises a set of characters enclosed by the rectangular brackets, [and],

but it matches a single character in the class. The pattern [abd] is character class, and it matches a single character – an a,b or d.

Examples:

$ls chap0[124] Matches chap01, chap02, chap04 and lists if found.

$ ls chap[x-z] Matches chapx, chapy, chapz and lists if found.

You can negate a character class to reverse a matching criteria. For example,

- To match all filenames with a single-character extension but not the .c ot .o files,
  use *.[!co]
- To match all filenames that don't begin with an alphabetic character,
  use [!a-zA-Z]*

Matching totally dissimilar patterns

This feature is not available in the Bourne shell. To copy all the C and Java source programs from another directory, we can delimit the patterns with a comma and then put curly braces around them.

$ cp $HOME/prog_sources/*.{c,java} .

The Bourne shell requires two separate invocations of cp to do this job.

$ cp /home/srm/{project,html,scripts/* .

The above command copies all files from three directories (project, html and scripts) to the current directory.

3. Escaping and Quoting

Escaping is providing a \ (backslash) before the wild-card to remove (escape) its special meaning.

For instance, if we have a file whose filename is chap* (Remember a file in UNIX can be names with virtually any character except the / and null), to remove the file, it is dangerous to give command as *rm chap**, as it will remove all files beginning with chap. Hence to suppress the special meaning of *, use the command *rm chap**

To list the contents of the file chap0[1-3], use

$ cat chap0\[1-3\]

A filename can contain a whitespace character also. Hence to remove a file named

My Documend.doc, which has a space embedded, a similar reasoning should be followed:

$ rm My\ Document.doc

Quoting is enclosing the wild-card, or even the entire pattern, within quotes. Anything within these quotes (barring a few exceptions) are left alone by the shell and not interpreted.

When a command argument is enclosed in quotes, the meanings of all enclosed special characters are turned off.

Examples:

$ rm 'chap*' Removes fil *chap**

$ rm "My Document.doc" Removes file *My Document.doc*

4. Redirection : The three standard files

The shell associates three files with the terminal – two for display and one for the keyboard. These files are streams of characters which many commands see as input and output. When a user logs in, the shell makes available three files representing three streams. Each stream is associated with a default device:
Standard input: The file (stream) representing input, connected to the keyboard.
Standard output: The file (stream) representing output, connected to the display.
Standard error: The file (stream) representing error messages that emanate from the command or shell, connected to the display.

The standard input can represent three input sources:
The keyboard, the default source.
A file using redirection with the < symbol.
Another program using a pipeline.

The standard output can represent three possible destinations:
The terminal, the default destination.
A file using the redirection symbols > and >>.
As input to another program using a pipeline.

A file is opened by referring to its pathname, but subsequent read and write operations identify the file by a unique number called a file descriptor. The kernel maintains a table of file descriptors for every process running in the system. The first three slots are generally allocated to the three standard streams as,

0 – Standard input
1 – Standard output
2 – Standard error

These descriptors are implicitly prefixed to the redirection symbols.
Examples:
Assuming file2 doesn't exist, the following command redirects the standard output to file *myOutput* and the standard error to file *myError*.

```
$ ls –l file1 file2 1>myOutput 2>myError
```

To redirect both standard output and standard error to a single file use:

```
$ ls –l file1 file2 1>| myOutput 2>| myError OR
$ ls –l file1 file2 1> myOutput 2>& 1
```

5. Filters: Using both standard input and standard output

UNIX commands can be grouped into four categories viz.,

1. Directory-oriented commands like mkdir, rmdir and cd, and basic file handling commands like cp, mv and rm use neither standard input nor standard output.
2. Commands like ls, pwd, who etc. don't read standard input but they write to standard output.
3. Commands like lp that read standard input but don't write to standard output.

4. Commands like cat, wc, cmp etc. that use both standard input and standard output.

Commands in the fourth category are called filters. Note that filters can also read directly from files whose names are provided as arguments.
Example: To perform arithmetic calculations that are specified as expressions in input file calc.txt and redirect the output to a file result.txt, use

```
$ bc < calc.txt   > result.txt
```

6. /dev/null and /dev/tty : Two special files

/dev/null: If you would like to execute a command but don't like to see its contents on the screen, you may wish to redirect the output to a file called /dev/null. It is a special file that can accept any stream without growing in size. It's size is always zero.

/dev/tty: This file indicates one's terminal. In a shell script, if you wish to redirect the output of some select statements explicitly to the terminal. In such cases you can redirect these explicitly to /dev/tty inside the script.

7. Pipes

With piping, the output of a command can be used as input (piped) to a subsequent command.

```
$ command1 | command2
```

Output from command1 is piped into input for command2.

This is equivalent to, but more efficient than:

```
$ command1 > temp
$ command2 < temp
$ rm temp
```

Examples

```
$ ls -al | more
$ who | sort | lpr
```

When a command needs to be ignorant of its source

If we wish to find total size of all C programs contained in the working directory, we can use the command,

```
$ wc –c *.c
```

However, it also shows the usage for each file(size of each file). We are not interested in individual statistics, but a single figure representing the total size. To be able to do that, we must make wc ignorant of its input source. We can do that by feeding the concatenated output stream of all the .c files to wc –c as its input:

```
$ cat *.c | wc –c
```

8. Creating a tee

tee is an external command that handles a character stream by duplicating its input. It saves one copy in a file and writes the other to standard output. It is also a filter and hence can be placed anywhere in a pipeline.

Example: The following command sequence uses tee to display the output of who and saves this output in a file as well.

```
$ who | tee users.lst
```

9. Command substitution

The shell enables the connecting of two commands in yet another way. While a pipe enables a command to obtain its standard input from the standard output of another command, the shell enables one or more command arguments to be obtained from the standard output of another command. This feature is called command substitution.

Example:

```
$ echo Current date and time is `date`
```

Observe the use of backquotes around date in the above command. Here the output of the command execution of date is taken as argument of echo. The shell executes the enclosed command and replaces the enclosed command line with the output of the command.

Similarly the following command displays the total number of files in the working directory.

```
$ echo “There are `ls | wc –l` files in the current directory”
```

Observe the use of double quotes around the argument of echo. If you use single quotes, the backquote is not interpreted by the shell if enclosed in single quotes.

10. Shell variables

Environmental variables are used to provide information to the programs you use. You can have both global environment and local shell variables. Global environment variables are set by your login shell and new programs and shells inherit the environment of their parent shell. Local shell variables are used only by that shell and are not passed on to other processes. A child process cannot pass a variable back to its parent process.

To declare a local shell variable we use the form *variable=value* (no spaces around =) and its evaluation requires the $ as a prefix to the variable.

Example:

```
$  count=5
$  echo $count
5
```

A variable can be removed with **unset** and protected from reassignment by **readonly**. Both are shell internal commands.

Note: In C shell, we use **set** statement to set variables. Here, there either has to be whitespace on both sides of the = or none at all.

```
$ set count=5
$ set size = 10
```

Uses of local shell variables

1. Setting pathnames: If a pathname is used several times in a script, we can assign it to a variable and use it as an argument to any command.
2. Using command substitution: We can assign the result of execution of a command to a variable. The command to be executed must be enclosed in backquotes.
3. Concatenating variables and strings: Two variables can be concatenated to form a new variable.

   Example:
   ```
   $ base=foo ; ext=.c
   $ file=$base$ext
   $ echo $file      // prints foo.c
   ```

Conclusion

In this chapter we saw the major interpretive features of the shell. The following is a summary of activities that the shell performs when a command line is encountered at the prompt.

- Parsing: The shell first breaks up the command line into words using spaces and tabs as delimiters, unless quoted. All consecutive occurrences of a space or tab are replaced with a single space.
- Variable evaluation: All $-prefixed strings are evaluated as variables, unless quoted or escaped.
- Command substitution: Any command surrounded by backquotes is executed by the shell, which then replaces the standard output of the command into the command line.
- Redirection: The shell then looks for the characters >, < and >> to open the files they point to.
- Wild-card interpretation: The shell then scans the command line for wild-cards (the characters *, ?, [and]). Any word containing a wild-card is replaced by a sorted list of filenames that match the pattern. The list of these filenames then forms the arguments to the command.
- PATH evaluation: It finally looks for the PATH variable to determine the sequence of directories it has to search in order to find the associated binary.

Chapter 7. The Process

Introduction

A process is an OS abstraction that enables us to look at files and programs as their time image. This chapter discusses processes, the mechanism of creating a process, different states of a process and also the ps command with its different options. A discussion on creating and controlling background jobs will be made next. We also look at three commands viz., at, batch and cron for scheduling jobs. This chapter also looks at nice command for specifying job priority, signals and time command for getting execution time usage statistics of a command.

Objectives

- Process Basics
- ps: Process Status
- Mechanism of Process Creation
- Internal and External Commands
- Process States and Zombies
- Background Jobs
- nice: Assigning execution priority
- Processes and Signals
- job Control
- at and batch: Execute Later
- cron command: Running Jobs Periodically
- time: Timing Usage Statistics at process runtime

1. Process Basics

UNIX is a multiuser and multitasking operating system. *Multiuser* means that several people can use the computer system simultaneously (unlike a single-user operating system, such as MS-DOS). *Multitasking* means that UNIX, like Windows NT, can work on several tasks concurrently; it can begin work on one task and take up another before the first task is finished.

When you execute a program on your UNIX system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system. Stated in other words, a process is created. A process is a program in execution. A process is said to be born when the program starts execution and remains alive as long as the program is active. After execution is complete, the process is said to die.

The kernel is responsible for the management of the processes. It determines the time and priorities that are allocated to processes so that more than one process can share the CPU resources.

Just as files have attributes, so have processes. These attributes are maintained by the kernel in a data structure known as process table. Two important

attributes of a process are:

1. The Process-Id (PID): Each process is uniquely identified by a unique integer called the PID, that is allocated by the kernel when the process is born. The PID can be used to control a process.
2. The Parent PID (PPID): The PID of the parent is available as a process attribute.

There are three types of processes viz.,

1. Interactive: Initiated by a shell and running in the foreground or background
2. batch: Typically a series of processes scheduled for execution at a specified point in time
3. daemon: Typically initiated at boot time to perform operating system functions on demand, such as LPD, NFS, and DNS

The Shell Process

As soon as you log in, a process is set up by the kernel. This process represents the login shell, which can be either sh(Bourne Shell), ksh(korn Shell), bash(Bourne Again Shell) or csh(C Shell).

Parents and Children

When you enter an external command at the prompt, the shell acts as the parent process, which in turn starts the process representing the command entered. Since every parent has a parent, the ultimate ancestry of any process can be traced back to the first process (PID 0) that is set up when the system is booted. It is analogous to the root directory of the file system. A process can have only one parent. However, a process can spawn multiple child processes.

Wait or not Wait?

A parent process can have two approaches for its child:

- It may wait for the child to die so that it can spawn the next process. The death of the child is intimated to the parent by the kernel. Shell is an example of a parent that waits for the child to terminate. However, the shell can be told not to wait for the child to terminate.
- It may not wait for the child to terminate and may continue to spawn other processes. init process is an example of such a parent process.

2. ps: Process Status

Because processes are so important to getting things done, UNIX has several commands that enable you to examine processes and modify their state. The most frequently used command is ps, which prints out the process status for processes running on your system. Each system has a slightly different version of the ps command, but there are two main variants, the System V version (POSIX) and the Berkeley version. The following table shows the options available with ps command.

POSIX	BSD	Significance
-f	f	Full listing showing PPID of each process
-e or –A	aux	All processes (user and system) processes
-u *user*	U *user*	Processes of user *user only*
-a		Processes of all users excluding processes not associated with terminal
-l	l	Long listing showing memory related information
-t *term*	t *term*	Processes running on the terminal *term*

Examples

$ **ps**

```
PID  TTY   TIME    CMD
4245 pts/7 00:00:00 bash
5314 pts/7 00:00:00 ps
```

The output shows the header specifying the PID, the terminal (TTY), the cumulative processor time (TIME) that has been consumed since the process was started, and the process name (CMD).

```
$ ps -f
UID   PID   PPID  C STIME    TTY   TIME COMMAND
root   14931 136   0 08:37:48 ttys0 0:00 rlogind
sartin 14932 14931 0 08:37:50 ttys0 0:00 -sh
sartin 15339 14932 7 16:32:29 ttys0 0:00 ps –f
```

The header includes the following information:

UID – Login name of the user
PID – Process ID
PPID – Parent process ID
C – An index of recent processor utilization, used by kernel for scheduling
STIME – Starting time of the process in hours, minutes and seconds
TTY – Terminal ID number
TIME – Cumulative CPU time consumed by the process
CMD – The name of the command being executed

System processes (-e or –A)

Apart from the processes a user generates, a number of system processes keep running all the time. Most of them are not associated with any controlling terminal.

They are spawned during system startup and some of them start when the system goes into multiuser mode. These processes are known as daemons because they are called without a specific request from a user. To list them use,

```
$ ps –e
PID      TTY   TIME    CMD
```

```
0       ?           0:34    sched
1       ?        41:55 init
23274 Console 0:03          sh
272     ?           2:47    cron
7015    term/12 20:04 vi
```

3. Mechanism of Process Creation

There are three distinct phases in the creation of a process and uses three important system calls viz., fork, *exec*, and wait. The three phases are discussed below:

- Fork: A process in UNIX is created with the fork system call, which creates a copy of the process that invokes it. The process image is identical to that of the calling process, except for a few parameters like the PID. The child gets a new PID.
- Exec: The forked child overwrites its own image with the code and data of the new program. This mechanism is called exec, and the child process is said to *exec* a new program, using one of the family of exec system calls. The PID and PPID of the exec'd process remain unchanged.
- Wait: The parent then executes the wait system call to *wait* for the child to complete. It picks up the exit status of the child and continues with its other functions. Note that a parent need not decide to wait for the child to terminate.

To get a better idea of this, let us explain with an example. When you enter ls to look at the contents of your current working directory, UNIX does a series of things to create an environment for ls and the run it:

- The shell has UNIX perform a fork. This creates a new process that the shell will use to run the ls program.
- The shell has UNIX perform an exec of the ls program. This replaces the shell program and data with the program and data for ls and then starts running that new program.
- The ls program is loaded into the new process context, replacing the text and data of the shell.
- The ls program performs its task, listing the contents of the current directory. In the meanwhile, the shell executes wait system call for ls to complete.

When a process is forked, the child has a different PID and PPID from its parent. However, it inherits most of the attributes of the parent. The important attributes that are inherited are:

- User name of the real and effective user (RUID and EUID): the owner of the process. The real owner is the user issuing the command, the effective user is the one determining access to system resources. RUID and EUID are usually the same, and the process has the same access rights the issuing user would have.
- Real and effective group owner (RGID and EGID): The real group owner of a process is the primary group of the user who started the process. The

effective group owner is usually the same, except when SGID access mode has been applied to a file.
- The current directory from where the process was run.
- The file descriptors of all files opened by the parent process.
- Environment variables like HOME, PATH.

The inheritance here means that the child has its own copy of these parameters and thus can alter the environment it has inherited. But the modified environment is not available to the parent process.

How the Shell is created?

- When the system moves to multiuser mode, **init** forks and execs a **getty** for every active communication port.
- Each one of these **getty**'s prints the login prompt on the respective terminal and then goes off to sleep.
- When a user tries to log in, **getty** wakes up and fork-execs the **login** program to verify login name and password entered.
- On successful login, **login** for-execs the process representing the login shell.
- **init** goes off to sleep, waiting for the children to terminate. The processes **getty** and **login** overlay themselves.
- When the user logs out, it is intimated to **init**, which then wakes up and spawns another **getty** for that line to monitor the next login.

4. Internal and External Commands

From the process viewpoint, the shell recognizes three types of commands:

1. External commands: Commonly used commands like **cat**, **ls** etc. The shell creates a process for each of these commands while remaining their parent.
2. Shell scripts: The shell executes these scripts by spawning another shell, which then executes the commands listed in the script. The child shell becomes the parent of the commands that feature in the shell.
3. Internal commands: When an internal command is entered, it is directly executed by the shell. Similarly, variable assignment like x=5, doesn't generate a process either.

Note: Because the child process inherits the current working directory from its parent as one of the environmental parameters, it is necessary for the cd command not to spawn a child to achieve a change of directory. If this is allowed, after the child dies, control would revert to the parent and the original directory would be restored. Hence, cd is implemented as an internal command.

5. Process States and Zombies

At any instance of time, a process is in a particular state. A process after creation

is in the *runnable* state. Once it starts running, it is in the *running* state. When a process requests for a resource (like disk I/O), it may have to wait. The process is said to be in *waiting* or *sleeping* state. A process can also be *suspended* by pressing a key (usually *Ctrl-z).*

When a process terminates, the kernel performs clean-up, assigns any children of the exiting process to be adopted by **init**, and sends the death of a child signal to the parent process, and converts the process into the zombie state.

A process in zombie state is not alive; it does not use any resources nor does any work. But it is not allowed to die until the exit is acknowledged by the parent process.

It is possible for the parent itself to die before the child dies. In such case, the child becomes an **orphan** and the kernel makes **init** the parent of the orphan. When this adopted child dies, **init** waits for its death.

6. Running Jobs in Background

The basic idea of a background job is simple. It's a program that can run without prompts or other manual interaction and can run in parallel with other active processes.

Interactive processes are initialized and controlled through a terminal session. In other words, there has to be someone connected to the system to start these processes; they are not started automatically as part of the system functions. These processes can run in the foreground, occupying the terminal that started the program, and you can't start other applications as long as this process is running in the foreground.

There are two ways of starting a job in the background – with the shell's & operator and the **nohup** command.

&: No Logging out

Ordinarily, when the shell runs a command for you, it waits until the command is completed. During this time, you cannot communicate with the shell. You can run a command that takes a long time to finish as a background job, so that you can be doing something else. To do this, use the & symbol at the end of the command line to direct the shell to execute the command in the background.

```
$ sort –o emp.dat emp.dat &
[1] 1413                    The job's PID
```

Note:

1. Observe that the shell acknowledges the background command with two numbers. First number [1] is the *job ID* of this command. The other number 1413 is the PID.
2. When you specify a command line in a pipeline to run in the background, all the commands are run in the background, not just the last command.
3. The shell remains the parent of the background process.

nohup: Log out Safely

A background job executed using & operator ceases to run when a user logs out. This is because, when you logout, the shell is killed and hence its children are also killed. The UNIX system provides nohup statement which when prefixed to a command, permits execution of the process even after the user has logged out. You must use the & with it as well.

The syntax for the nohup command is as follows:

```
nohup command-string [input-file] output-file &
```

If you try to run a command with nohup and haven't redirected the standard error, UNIX automatically places any error messages in a file named nohup.out in the directory from which the command was run.

In the following command, the sorted file and any error messages are placed in the file nohup.out.

```
$ nohup sort sales.dat &
1252
Sending output to nohup.out
```

Note that the shell has returned the PID (1252) of the process.
When the user logs out, the child turns into an orphan. The kernel handles such situations by reassigning the PPID of the orphan to the system's init process (PID 1) - the parent of all shells. When the user logs out, init takes over the parentage of any process run with nohup. In this way, you can kill a parent (the shell) without killing its child.

Additional Points

When you run a command in the background, the shell disconnects the standard input from the keyboard, but does not disconnect its standard output from the screen. So, output from the command, whenever it occurs, shows up on screen. It can be confusing if you are entering another command or using another program. Hence, make sure that both standard output and standard error are redirected suitably.

```
    $ find . –name "*.log" –print> log_file 2> err.dat &
OR  $ find . –name "*.log" –print> log_file 2> /dev/null &
```

Important:

1. You should relegate time-consuming or low-priority jobs to the background.
2. If you log out while a background job is running, it will be terminated.

7. nice: Job Execution with Low Priority

Processes in UNIX are sequentially assigned resources for execution. The kernel assigns the CPU to a process for a time slice; when the time elapses, the process

is places in a queue. How the execution is scheduled depends on the priority assigned to the process.

The *nice* command is used to control background process dispatch priority.

The idea behind *nice* is that background jobs should demand less attention from the system than interactive processes.
Background jobs execute without a terminal attached and are usually run in the background for two reasons:

1. the job is expected to take a relatively long time to finish, and
2. the job's results are not needed immediately.

Interactive processes, however, are usually shells where the speed of execution is critical because it directly affects the system's apparent response time. It would therefore be nice for everyone (others as well as you) to let interactive processes have priority over background work.
nice values are system dependent and typically range from 1 to 19.

A high *nice* value implies a lower priority. A program with a high nice number is friendly to other programs, other users and the system; it is not an important job. The lower the nice number, the more important a job is and the more resources it will take without sharing them.
Example:

```
        $ nice wc –l hugefile.txt
OR      $ nice wc –l hugefile.txt &
```

The default nice value is set to 10.
We can specify the nice value explicitly with –n *number* option where *number* is an offset to the default. If the –n *number* argument is present, the priority is incremented by that amount up to a limit of 20.
Example: `$ nice –n 5 wc –l hugefile.txt &`

8. Killing Processes with Signals

When you execute a command, one thing to keep in mind is that commands do not run in a vacuum. Many things can happen during a command execution that are not under the control of the command. The user of the command may press the interrupt key or send a kill command to the process, or the controlling terminal may become disconnected from the system. In UNIX, any of these events can cause a **signal** to be sent to the process. The default action when a process receives a signal is to terminate.

When a process ends normally, the program returns its *exit status* to the parent. This exit status is a number returned by the program providing the results of the program's execution.
Sometimes, you want or need to terminate a process.
The following are some reasons for stopping a process:

- It’s using too much CPU time.
- It’s running too long without producing the expected output.

- It's producing too much output to the screen or to a disk file.
- It appears to have locked a terminal or some other session.
- It's using the wrong files for input or output because of an operator or programming error.
- It's no longer useful.

If the process to be stopped is a background process, use the kill command to get out of these situations. To stop a command that isn't in the background, press <ctrl-c>.

To use kill, use either of these forms:
kill PID(s) OR kill –s NUMBER PID(s)
To kill a process whose PID is 123 use,
$ kill 123
To kill several processes whose PIDs are 123, 342, and 73 use,
$ kill 123 342 73

Issuing the kill command sends a signal to a process. The default signal is SIGTERM signal (15). UNIX programs can send or receive more than 20 signals, each of which is represented by a number. (Use kill –l to list all signal names and numbers)

If the process ignores the signal SIGTERM, you can kill it with SIGKILL signal (9) as,
$ kill -9 123 OR $ kill –s KILL 123
The system variable $! stores the PID of the last background job. You can kill the last background job without knowing its PID by specifying $ kill $!
Note: You can kill only those processes that you own; You can't kill processes of other users. To kill all background jobs, enter kill 0.

9. Job Control

A job is a name given to a group of processes that is typically created by piping a series of commands using pipeline character. You can use job control facilities to manipulate jobs. You can use job control facilities to,

1. Relegate a job to the background (bg)
2. Bring it back to the foreground (fg)
3. List the active jobs (jobs)
4. Suspend a foreground job (*[Ctrl-z]*)
5. Kill a job (kill)

The following examples demonstrate the different job control facilities.
Assume a process is taking a long time. You can suspend it by pressing *[Ctrl-z].*
[1] + Suspended wc –l hugefile.txt
A suspended job is not terminated. You can now relegate it to background by,
$ bg
You can start more jobs in the background any time:
$ sort employee.dat > sortedlist.dat &

[2] 530
$ grep ‘director’ emp.dat &
[3] 540
You can see a listing of these jobs using jobs command,
$ jobs
[3] + Running grep ‘director’ emp.dat &
[2] - Running sort employee.dat > sortedlist.dat &
[1] Suspended wc –l hugefile.txt
You can bring a job to foreground using fg %jobno OR fg %jobname as,
$ fg %2 OR $ fg %sort

10. at And batch: Execute Later

UNIX provides facilities to schedule a job to run at a specified time of day. If the system load varies greatly throughout the day, it makes sense to schedule less important jobs at a time when the system load is low. The at and batch commands make such job scheduling possible.

at: One-Time Execution

To schedule one or more commands for a specified time, use the at command. With this command, you can specify a time, a date, or both.
For example,
$ at 14:23 Friday
at> lp /usr/sales/reports/*
at> echo “Files printed, Boss!” | mail -s”Job done” boss
[Ctrl-d]
commands will be executed using /usr/bin/bash
job 1041198880.a at Fri Oct 12 14:23:00 2007

The above job prints all files in the directory /usr/sales/reports and sends a user named boss some mail announcing that the print job was done.
All at jobs go into a queue known as at queue.at shows the job number, the date and time of scheduled execution. This job number is derived from the number of seconds elapsed since the Epoch. A user should remember this job number to control the job.

$ at 1 pm today
at> echo “^G^GLunch with Director at 1 PM^G^G” > /dev/term/43

The above job will display the following message on your screen (/dev/term/43) at 1:00 PM, along with two beeps(^G^G).
Lunch with Director at 1 PM

To see which jobs you scheduled with at, enter at -l. Working with the preceding examples, you may see the following results:

job 756603300.a at Tue Sep 11 01:00:00 2007
job 756604200.a at Fri Sep 14 14:23:00 2007

The following forms show some of the keywords and operations permissible with at command:

at hh:mm Schedules job at the hour (*hh*) and minute (*mm*) specified, using a 24-hour clock

at hh:mm month day year Schedules job at the hour (*hh*), minute (*mm*), month, day, and year specified

at -l Lists scheduled jobs

at now +count time-units Schedules the job right now plus *count* number of *timeunits*; time units can be minutes, hours, days, or weeks

at –r job_id Cancels the job with the job number matching *job_id*

batch: Execute in Batch Queue

The batch command lets the operating system decide an appropriate time to run a process. When you schedule a job with batch, UNIX starts and works on the process whenever the system load isn't too great.

To sort a collection of files, print the results, and notify the user named boss that the job is done, enter the following commands:

```
$ batch
sort /usr/sales/reports/* | lp
echo “Files printed, Boss!” | mailx -s”Job done” boss
```

The system returns the following response:

```
job 7789001234.b at Fri Sep 7 11:43:09 2007
```

The date and time listed are the date and time you pressed <Ctrl-d> to complete the batch command. When the job is complete, check your mail; anything that the commands normally display is mailed to you. Note that any job scheduled with batch command goes into a special at queue.

11. cron: Running jobs periodically

cron program is a daemon which is responsible for running repetitive tasks on a regular schedule. It is a perfect tool for running system administration tasks such as backup and system logfile maintenance. It can also be useful for ordinary users to schedule regular tasks including calendar reminders and report generation.

Both *at* and *batch* schedule commands on a one-time basis. To schedule commands or processes on a regular basis, you use the cron (short for *chronograph*) program. You specify the times and dates you want to run a command in crontab files. Times can be specified in terms of minutes, hours, days of the month, months of the year, or days of the week.

cron is listed in a shell script as one of the commands to run during a system boot-up sequence. Individual users don't have permission to run cron directly.

If there's nothing to do, cron "goes to sleep" and becomes inactive; it "wakes up" every minute, however, to see if there are commands to run.
cron looks for instructions to be performed in a control file in
/var/spool/cron/crontabs
After executing them, it goes back to sleep, only to wake up the next minute.

To a create a crontab file,
First use an editor to create a crontab file say cron.txt
Next use crontab command to place the file in the directory containing crontab files. crontab will create a file with filename same as user name and places it in /var/spool/cron/crontabs directory.

Alternately you can use crontab with –e option.

You can see the contents of your crontab file with crontab –l and remove them with crontab –r.

The cron system is managed by the cron daemon. It gets information about which programs and when they should run from the system's and users' crontab entries. The crontab files are stored in the file /var/spool/cron/crontabs/<user> where <user> is the login-id of the user. Only the root user has access to the system crontabs, while each user should only have access to his own crontabs.

A typical entry in crontab file

A typical entry in the crontab file of a user will have the following format.
minute hour day-of-month month-of-year day-of-week command

where, Time-Field Options are as follows:

Field	Range	
minute	00 through 59	Number of minutes after the hour
hour	00 through 23 (midnight is 00)	
day-of-month	01 through 31	
month-of-year	01 through 12	
day-of-week	01 through 07 (Monday is 01, Sunday is 07)	

The first five fields are time option fields. You must specify all five of these fields. Use an asterisk (*) in a field if you want to ignore that field.

Examples:
00-10 17 * 3.6.9.12 5 find / -newer .last_time –print >backuplist
In the above entry, the find command will be executed every minute in the first 10 minutes after 5 p.m. every Friday of the months March, June, September and December of every year.

30 07 * * 01 sort /usr/wwr/sales/weekly |mail -s"Weekly Sales" srm
In the above entry, the sort command will be executed with /usr/www/sales/weekly as argument and the output is mailed to a user named srm at 7:30 a.m. each Monday.

12. time: Timing Processes

The time command executes the specified command and displays the time usage on the terminal.
Example: You can find out the time taken to perform a sorting operation by preceding the sort command with time.

```
$ time sort employee.dat > sortedlist.dat
real    0m29.811s
user    0m1.370s
sys     0m9.990s
```

where,
the *real* time is the clock elapsed from the invocation of the command until its termination.
the *user* time shows the time spent by the program in executing itself.
the *sys* time indicates the time used by the kernel in doing work on behalf of a user process.
The sum of user time and sys time actually represents the CPU time. This could be significantly less than the real time on a heavily loaded system.

Conclusion

In this chapter, we saw an important abstraction of the UNIX operating system viz., processes. We also saw the mechanism of process creation, the attributes inherited by the child from the parent process as well as the shell's behavior when it encounters internal commands, external commands and shell scripts. This chapter also discussed background jobs, creation and controlling jobs as well as controlling processes using signals. We finally described three commands viz., at, batch and cron for process scheduling, with a discussion of time command for obtaining time usage statistics of process execution.

Chapter 8. Customizing the Environment

Introduction

The UNIX environment can be highly customized by manipulating the settings of the shell. Commands can be made to change their default behavior, environment variables can be redefined, the initialization scripts can be altered to obtain a required shell environment. This chapter discusses different ways and approaches for customizing the environment.

Objectives

- The Shell
- Environment Variables
- Common Environment Variables
- Command Aliases (bash and korn)
- Command History Facility (bash and korn)
- In-Line Command Editing (bash and korn)
- Miscellaneous Features (bash and korn)
- The Initialization Scripts

The Shell

The UNIX shell is both an interpreter as well as a scripting language. An interactive shell turns noninteractive when it executes a script.

Bourne Shell – This shell was developed by Steve Bourne. It is the original UNIX shell. It has strong programming features, but it is a weak interpreter.

C Shell – This shell was developed by Bill Joy. It has improved interpretive features, but it wasn't suitable for programming.

Korn Shell – This shell was developed by David Korn. It combines best features of the bourne and C shells. It has features like aliases, command history. But it lacks some features of the C shell.

Bash Shell – This was developed by GNU. It can be considered as a superset that combined the features of Korn and C Shells. More importantly, it conforms to POSIX shell specification.

Environment Variables

We already mentioned a couple of environment variables, such as PATH and HOME. Until now, we only saw examples in which they serve a certain purpose to the shell. But there are many other UNIX utilities that need information about you in order to do a good job.

What other information do programs need apart from paths and home directories? A lot of programs want to know about the kind of terminal you are using; this information is stored in the TERM variable. The shell you are using is stored in the SHELL variable, the operating system type in OS and so on. A list of all variables currently defined for your session can be viewed entering the **env** command.

The environment variables are managed by the shell. As opposed to regular shell variables, environment variables are inherited by any program you start, including another shell. New processes are assigned a copy of these variables, which they can read, modify and pass on in turn to their own child processes.

The set statement display all variables available in the current shell, but env command displays only environment variables. Note than env is an external command and runs in a child process.

There is nothing special about the environment variable names. The convention is to use uppercase letters for naming one.

The Common Environment Variables

The following table shows some of the common environment variables.

Variable name	Stored information
HISTSIZE	size of the shell history file in number of lines
HOME	path to your home directory
HOSTNAME	local host name
LOGNAME	login name
MAIL	location of your incoming mail folder
MANPATH	paths to search for man pages
PATH	search paths for commands
PS1	primary prompt
PS2	secondary prompt
PWD	present working directory
SHELL	current shell
TERM	terminal type
UID	user ID
USER	Login name of user
MAILCHECK	Mail checking interval for incoming mail
CDPATH	List of directories searched by cd when used with a non-absolute pathname

We will now describe some of the more common ones.

The command search path (PATH): The PATH variable instructs the shell about the route it should follow to locate any executable command.
Your home directory (HOME): When you log in, UNIX normally places you in a directory named after your login name. This is called the home directory or login directory. The home directory for a user is set by the system administrator while creating users (using useradd command).
mailbox location and checking (MAIL and MAILCHECK): The incoming mails for a user are generally stored at /var/mail or /var/spool/mail and this location is available in the environment variable MAIL. MAILCHECK determines how often the shell checks the file for arrival of new mail.
The prompt strings (PS1, PS2): The prompt that you normally see (the $ prompt) is the shell's primary prompt specified by PS1. PS2 specifies the secondary prompt (>). You can change the prompt by assigning a new value to these environment variables.
Shell used by the commands with shell escapes (SHELL): This environment variable specifies the login shell as well as the shell that interprets the command if preceded with a shell escape.

Variables used in Bash and Korn

The Bash and korn prompt can do much more than displaying such simple information as your user name, the name of your machine and some indication about the present working directory. Some examples are demonstrated next.

$ PS1='[PWD] '
[/home/srm] cd progs
[/home/srm/progs] _

Bash and Korn also support a *history* facility that treats a previous command as an *event* and associates it with a number. This event number is represented as !.

$ PS1='[!] '
[42] _

$ PS1='[! $PWD] '
[42 /home/srm/progs] _

$ PS1="\h> "
saturn> _

// Host name of the machine

Aliases

Bash and korn support the use of aliases that let you assign shorthand names to frequently used commands. Aliases are defined using the alias command. Here are some typical aliases that one may like to use:

```
alias lx='/usr/bin/ls -lt'
alias l='/usr/bin/ls -l'
```

You can also use aliasing to redefine an existing command so it is always invoked with certain options. For example:

```
alias cp="cp –i"
alias rm="rm –i"
```

Note that to execute the original external command, you have to precede the command with a \. This means that you have to use \cp file1 file2 to override the alias.

The alias command with a argument displays its alias definition, if defined. The same command without any arguments displays all aliases and to unset an alias use unalias statement. To unset the alias cp, use unalias cp

Command History

Bash and Korn support a history feature that treats a previous command as an event and associates it with an event number. Using this number you can recall previous commands, edit them if required and reexecute them.

The history command displays the history list showing the event number of every previously executed command. With bash, the complete history list is displayed, while with korn, the last 16 commands. You can specify a numeric argument to specify the number of previous commands to display, as in, history 5 (in bash) or history -5 (korn).

By default, bash stores all previous commands in $HOME/.bash_history and korn stores them in $HOME/.sh_history. When a command is entered and executed, it is appended to the list maintained in the file.

Accessing previous commands by Event Numbers (! and r)

The ! symbol (r in korn) is used to repeat previous commands. The following examples demonstrate the use of this symbol with corresponding description.

$!38 The command with event number 38 is displayed and executed (Use r 38 in korn)

$!38:p The command is displayed. You can edit and execute it

$!! Repeats previous command (Use r in korn)

$!-2 Executes command prior to the previous one (r -2 in korn)

Executing previous commands by Context

When you don't remember the event number of a command but know that the command started with a specific letter of a string, you can use the history facility with context.

Example: **$!v** Repeats the last command beginning with v (**r v** in korn)

Substitution in previous commands

If you wish to execute a previous command after some changes, you can substitute the old string with new one by substitution.

If a previous command cp progs/*.doc backup is to be executed again with doc replaced with txt,

$!cp:s/doc/txt in bash

$ r cp doc=txt in korn

$_ is a shorthand feature to represent the directory used by the previous command.

$ mkdir progs

Now, instead of using cd progs, you can use,

$ cd $_

The History Variables

The command history will be maintained in default history files viz.,

.bash_history in Bash

.sh_history in Korn

Variable HISTFILE determines the filename that saves the history list. Bash uses two variables HISTSIZE for setting the size of the history list in memory and HISTFILESIZE for setting the size of disk file. Korn uses HISTSIZE for both the purposes.

In-Line Command Editing

One of the most attractive aspects of bash and korn shells is their treatment of command line editing. In addition to viewing your previous commands and reexecuting them, these shells let you edit your current command line, or any of the commands in your history list, using a special command line version of vi text editor. We have already seen the features of vi as a text editor and these features

can be used on the current command line, by making the following setting:

set –o vi

Command line editing features greatly enhance the value of the history list. You can use them to correct command line errors and to save time and effort in entering commands by modifying previous commands. It also makes it much easier to search through your command history list, because you can use the same search commands you use in vi.

Miscellaneous Features (bash and korn)

1. Using set –o

The set statement by default displays the variables in the current shell, but in Bash and Korn, it can make several environment settings with –o option.

File Overwriting(noclobber): The shell's > symbol overwrites (clobbers) an existing file, and o prevent such accidental overwriting, use the noclobber argument:

set –o noclobber

Now, if you redirect output of a command to an existing file, the shell will respond with a message that says it "cannot overwrite existing file" or "file already exists". To override this protection, use the | after the > as in,

head –n 5 emp.dat >| file1

Accidental Logging out (ignoreeof): The [Ctrl-d] key combination has the effect of terminating the standard input as well as logging out of the system. In case you accidentally pressed [Ctrl-d] twice while terminating the standard input, it will log you off! The ignoreeof keyword offers protection from accidental logging out:

set –o ignoreeof

But note that you can logout only by using exit command.

A set option is turned off with set +o *keyword*. To reverse the noclobber feature, use

set +o noclobber

2. Tilde Substitution

The ~ acts as a shorthand representation for the home directory. A configuration file like .profile that exists in the home directory can be referred to both as $HOME/.profile and ~/.profile.

You can also toggle between the directory you switched to most recently and your current directory. This is done with the ~- symbols (or simply -, a hyphen). For example, either of the following commands change to your previous directory:

cd ~- OR cd –

The Initialization Scripts

The effect of assigning values to variables, defining aliases and using set options is applicable only for the login session; they revert to their default values when the user logs out. To make them permanent, use certain startup scripts. The startup scripts are executed when the user logs in. The initialization scripts in different shells are listed below:

.profile (Bourne shell)
.profile and .kshrc (Korn shell)
.bash_profile (or .bash_login) and .bashrc (Bash)
.login and .cshrc (C shell)

The Profile

When logging into an interactive login shell, login will do the authentication, set the environment and start your shell. In the case of bash, the next step is reading the general profile from /etc, if that file exists. bash then looks for ~/.bash_profile, ~/.bash_login and ~/.profile, in that order, and reads and executes commands from the first one that exists and is readable. If none exists, /etc/bashrc is applied.

When a login shell exits, bash reads and executes commands from the file, ~/.bash_logout, if it exists.

The profile contains commands that are meant to be executed only once in a session. It can also be used to customize the operating environment to suit user requirements. Every time you change the profile file, you should either log out and log in again or You can execute it by using a special command (called dot).

```
$ . .profile
```

The rc File

Normally the profiles are executed only once, upon login. The rc files are designed to be executed every time a separate shell is created. There is no rc file in Bourne, but bash and korn use one. This file is defined by an environment variable BASH_ENV in Bash and ENV in Korn.

```
export BASH_ENV=$HOME/.bashrc
export ENV=$HOME/.kshrc
```

Korn automatically executes .kshrc during login if ENV is defined. Bash merely ensures that a sub-shell executes this file. If the login shell also has to execute this file then a separate entry must be added in the profile:

```
. ~/.bashrc
```

The rc file is used to define command aliases, variable settings, and shell options. Some sample entries of an rc file are

```
alias cp="cp –i"
alias rm="rm –i"
set –o noclobber
set –o ignoreeof
set –o vi
```

The rc file will be executed after the profile. However, if the BASH_ENV or ENV variables are not set, the shell executes only the profile.

Conclusion

In this chapter, we looked at the environment-related features of the shells, and found weaknesses in the Bourne shell. Knowledge of Bash and Korn only supplements your knowledge of Bourne and doesn’t take anything away. It is always advisable to use Bash or korn as your default login shell as it results in a more fruitful experience, with their rich features in the form of aliases, history features and in-line command editing features.

Chapter 14. Perl – The Mater Manipulator

Introduciton

The following sections tell you what Perl is, the variables and operators in perl, the string handling functions. The chapter also discusses file handling in perl as also the lists, arrays and associative arrays (hashes) that have made perl a popular scripting language. One or two lines of code in perl accomplish many lines of code in a high level language. We finally discuss writing subroutines in perl.

Objectives

- perl preliminaries
- The chop function
- Variables and Operators
- String handling functions
- Specifying filenames in a command line
- $_(Default Variable)
- $. (Current Line Number) and .. (The Range Operator)
- Lists and Arrays
- ARGV[]: Command Line Arguments
- foreach: Looping Through a List
- split: Splitting into a List or Array
- join: Joining a List
- dec2bin.pl: Converting a Decimal Number to Binary
- grep: Searching an Array for a Pattern
- Associative Arrays
- Regular Expressions and Substitution
- File Handling
- Subroutines
- Conclusion

1. Perl preliminaries

Perl: Perl stands for Practical Extraction and Reporting Language. The language was developed by Larry Wall. Perl is a popular programming language because of its powerful pattern matching capabilities, rich library of functions for arrays, lists and file handling. Perl is also a popular choice for developing CGI (Common Gateway Interface) scripts on the www (World Wide Web).

Perl is a simple yet useful programming language that provides the convenience of shell scripts and the power and flexibility of high-level programming languages. Perl programs are interpreted and executed directly, just as shell scripts are; however, they also contain control structures and operators similar to those found in the C programming language. This gives you the ability to write useful programs in a very
short time.

Perl is a freeware and can be obtained from http://www.perl.com or http://www.activestate.com (Perl interpreter for Windows).

A perl program runs in a special interpretive model; the entire script is compiled internally in memory before being executed. Script errors, if any, are generated before execution. Unlike awk, printing isn't perl's default action. Like C, all perl statements end with a semicolon. Perl statements can either be executed on command line with the –e option or placed in .pl files. In Perl, anytime a # character is recognized, the rest of the line is treated as a comment.

The following is a sample perl script.

```
#!/usr/bin/perl
# Script: sample.pl – Shows the use of variables
#
print("Enter your name: ");
$name=<STDIN>;
Print("Enter a temperature in Centigrade: ");
$centigrade=<STDIN>;
$fahr=$centigrade*9/5 + 32;
print "The temperature in Fahrenheit is $fahr\n";
print "Thank you $name for using this program."
```

There are two ways of running a perl script. One is to assign execute (x) permission on the script file and run it by specifying script filename (chmod +x filename). Other is to use perl interpreter at the command line followed by the script name. In the second case, we don't have to use the interpreter line viz., #!/usr/bin/perl.

2. The chop function

The chop function is used to remove the last character of a line or string. In the above program, the variable $name will contain the input entered as well as the newline character that was entered by the user. In order to remove the \n from the input variable, we use chop($name).

Example: chop($var); will remove the last character contained in the string specified by the variable *var*.

Note that you should use chop function whenever you read a line from the keyboard or a file unless you deliberately want to retain the newline character.

3. Variables and Operators

Perl variables have no type and need no initialization. However we need to precede the variable name with a $ for both variable initialization as well as evaluation.

Example:
```
$var=10;
print $var;
```

Some important points related to variables in perl are:

1. When a string is used for numeric computation or comparison, perl

converts it into a number.
2. If a variable is undefined, it is assumed to be a null string and a null string is numerically zero. Incrementing an uninitialized variable returns 1.
3. If the first character of a string is not numeric, the entire string becomes numerically equivalent to zero.
4. When Perl sees a string in the middle of an expression, it converts the string to an integer. To do this, it starts at the left of the string and continues until it sees a letter that is not a digit. Example: "12O34" is converted to the integer 12, not 12034.

Comparison Operators

Perl supports operators similar to C for performing numeric comparison. It also provides operators for performing string comparison, unlike C where we have to use either strcmp() or strcmpi() for string comparison. The are listed next.

Numeric comparison	String comparison
==	eq
!=	ne
>	gt
<	lt
>=	ge
<=	le

Concatenating and Repeating Strings

Perl provides three operators that operate on strings:

- The . operator, which joins two strings together;
- The x operator, which repeats a string; and
- The .= operator, which joins and then assigns.

The . operator joins the second operand to the first operand:
Example:

```
$a = “Info" . “sys";     # $a is now “Infosys"
$x=”microsoft”; $y=”.com”; $x=$x . $y;   # $x is now “microsoft.com”
```

This join operation is also known as string concatenation.

The x operator (the letter x) makes *n* copies of a string, where *n* is the value of the right operand:
Example:

```
$a = “R" x 5;            # $a is now “RRRRR"
```

The .= operator combines the operations of string concatenation and assignment:
Example:

```
$a = “VTU";
$a .= “ Belgaum";      # $a is now “VTU Belgaum"
```

4. String Handling Functions

Perl has all the string handling functions that you can think of. We list some of

the frequently used functions are:
length determines the length of its argument.
index(s1, s2) determines the position of a string **s2** within string **s1**.
substr(str,m,n) extracts a substring from a string **str**, **m** represents the starting point of extraction and **n** indicates the number of characters to be extracted.
uc(str) converts all the letters of str into uppercase.
ucfirst(str) converts first letter of all leading words into uppercase.
reverse(str) reverses the characters contained in string str.

5. Specifying Filenames in Command Line

Unlike awk, perl provides specific functions to open a file and perform I/O operations on it. We will look at them in a subsequent section. However, perl also supports special symbols that perform the same functionality. The diamond operator, <> is used for reading lines from a file. When you specify STDIN within the <>, a line is read from the standard input.
Example:
1. perl –e 'print while (<>)' sample.txt
2. perl –e 'print <>' sample.txt
In the first case, the file opening is implied and <> is used in scalar context (reading one line).
In the second case, the loop is also implied but <> is interpreted in list context (reading all lines).

The following script will print all Gupta's and Agarwal/Aggarwal's contained in a file (specified using an ERE) that is specified as a command line parameter along with the script name.

```
#!/usr/bin/perl
printf(%30s", "LIST OF EMPLOYEES\n");
while(<>) {
  print if /\bGupta|Ag+[ar][ar]wal/ ;
}
```

6. $_ : The Default Variable

perl assigns the line read from input to a special variable, $_, often called the default variable. chop, <> and pattern matching operate on $_ be default. It represents the last line read or the last pattern matched.

By default, any function that accepts a scalar variable can have its argument omitted. In this case, Perl uses $_, which is the default scalar variable. chop, <> and pattern matching operate on $_ by default, the reason why we did not specify it explicitly in the print statement in the previous script. The $_ is an important variable, which makes the perl script compact.

For example, instead of writing

```
$var = <STDIN>;
chop($var);
```

you can write,

chop(<STDIN>);

In this case, a line is read from standard input and assigned to default variable $_, of which the last character (in this case a \n) will be removed by the chop() function.

Note that you can reassign the value of $_, so that you can use the functions of perl without specifying either $_ or any variable name as argument.

7. $. (Current Line number) And .. (The range operator)

$. is the current line number. It is used to represent a line address and to select lines from anywhere.

Example:

perl –ne 'print if ($. < 4)' in.dat # is similar ro head –n 3 in.dat

perl –ne 'print if ($. > 7 && $. < 11)' in.dat # is similar to sed –n '8,10p'

.. is the range operator.

Example:

perl –ne 'print if (1..3)' in.dat # Prints lines 1 to 3 from in.dat

perl –ne 'print if (8..10)' in.dat # Prints lines 8 to 10 from in.dat

You can also use compound conditions for selecting multiple segments from a file.

Example: if ((1..2) || (13..15)) { print ;} # Prints lines 1 to 2 and 13 to 15

8. Lists and Arrays

Perl allows us to manipulate groups of values, known as lists or arrays. These lists can be assigned to special variables known as array variables, which can be processed in a variety of ways.

A list is a collection of scalar values enclosed in parentheses. The following is a simple example of a list:

(1, 5.3, "hello", 2)

This list contains four elements, each of which is a scalar value: the numbers 1 and 5.3, the string "hello", and the number 2.

To indicate a list with no elements, just specify the parentheses: ()

You can use different ways to form a list. Some of them are listed next.

- Lists can also contain scalar variables:
  (17, $var, "a string")
- A list element can also be an expression:
  (17, $var1 + $var2, 26 << 2)
- Scalar variables can also be replaced in strings:
  (17, "the answer is $var1")
- The following is a list created using the list range operator:
  (1..10)➔ same as (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
- The list range operator can be used to define part of a list:
  (2, 5..7, 11)
  The above list consists of five elements: the numbers 2, 5, 6, 7 and 11

Arrays

Perl allows you to store lists in special variables designed for that purpose. These variables are called array variables. Note that arrays in perl need not contain similar type of data. Also arrays in perl can dynamically grow or shrink at run time.

@array = (1, 2, 3); # Here, the list (1, 2, 3) is assigned to the array variable @array.

Perl uses @ and $ to distinguish array variables from scalar variables, the same name can be used in an array variable and in a scalar variable:

$var = 1;
@var = (11, 27.1, "a string");

Here, the name var is used in both the scalar variable $var and the array variable @var. These are two completely separate variables. You retrieve value of the scalar variable by specifying $var, and of that of array at index 1 as $var[1] respectively.

Following are some of the examples of arrays with their description.

x = 27; # list containing one element
@y = @x; # assign one array variable to another
@x = (2, 3, 4);
@y = (1, @x, 5); # the list (2, 3, 4) is substituted for @x, and the resulting list
(1, 2, 3, 4,5) is assigned to @y.

$len = @y; # When used as an *rvalue* of an assignment, @y evaluates to the
length of the array.
$last_index = $#y; # $# prefix to an array signifies the last index of the array.

9. ARGV[]: Command Line Arguments

The special array variable @ARGV is automatically defined to contain the strings entered on the command line when a Perl program is invoked. For example, if the program (test.pl):

```
#!/usr/bin/perl
print("The first argument is $ARGV[0]\n");
```

Then, entering the command

$ test.pl 1 2 3

produces the following output:

The first argument is 1

Note that $ARGV[0], the first element of the @ARGV array variable, does not contain the name of the program. This is a difference between Perl and C.

Modifying Array Contents

For deleting elements at the beginning or end of an array, perl uses the shift and pop functions. In that sense, array can be thought of both as a stack or a queue.
Example:

```
@list = (3..5, 9);
shift(@list);   # The 3 goes away, becomes 4 5 9
pop(@list);     # Removes last element, becomes 4 5
```

The unshift and push functions add elements to an array.

```
unshift(@list, 1..3);   # Adds 1, 2 and 3 –- 1 2 3 4 5
push(@list,9);          # Pushes 9 at end –- 1 2 3 4 5 9
```

The splice function can do everything that shift, pop, unshift and push can do. It uses upto four arguments to add or remove elements at any location in the array. The second argument is the offset from where the insertion or removal should begin. The third argument represents the number of elements to be removed. If it is 0, elements have to be added. The new replaced list is specified by the fourth argument (if present).

```
splice(@list, 5, 0, 6..8);   # Adds at 6th location, list becomes 1 2 3 4 5 6 7 8 9
splice(@list, 0, 2);  # Removes from beginning, list becomes 3 4 5 6 7 8 9
```

10. foreach: Looping Through a List

foreach construct is used to loop through a list. Its general form is,

```
foreach $var in (@arr) {
        statements
}
```

Example: To iterate through the command line arguments (that are specified as numbers) and find their square roots,

```
foreach $number (@ARGV) {
        print(“The square root of $number is ” .
sqrt($number) .  “\n”);
}
```

You can even use the following code segment for performing the same task. Here note the use of $_ as a default variable.

```
foreach (@ARGV) {
  print(“The square root of $_ is “ . sqrt() . “\”);
}
```

Another Example

```
#!/usr/bin/perl
@list = ("This", "is", "a", "list", "of", "words");
print("Here are the words in the list: \n");
foreach $temp (@list) {
print("$temp ");
}
print("\n");
```

Here, the loop defined by the foreach statement executes once for each element

in the list @list. The resulting output is
Here are the words in the list:
This is a list of words
The current element of the list being used as the counter is stored in a special scalar variable, which in this case is $temp. This variable is special because it is only defined for the statements inside the foreach loop.

perl has a for loop as well whose syntax similar to C.
Example:

```
for($i=0 ; $i < 3 ; $i++) { . . .
```

11. split: Splitting into a List or Array

There are two important array handling functions in perl that are very useful in CGI programming, viz., split and join.

split breaks up a line or expression into fields. These fields are assigned either to variables or an array.
Syntax:

```
($var1, $var2, $var3 …….  ) = split(/sep/, str);
@arr = split(/sep/, str);
```

It splits the string **str** on the pattern **sep**. Here **sep** can be a regular expression or a literal string. **str** is optional, and if absent, **$_** is used as default. The fields resulting from the split are assigned to a set of variables , or to an array.

12. join: Joining a List

It acts in an opposite manner to split. It combines all array elements in to a single string. It uses the delimiter as the first argument. The remaining arguments could be either an array name or a list of variables or strings to be joined.

```
$x = join(" ", "this", "is", "a", "sentence");    # $x becomes "this is a sentence".
@x = ("words","separated","by");
$y = join("::",@x,"colons");     #$y becomes "words::separated::by::colons".
```

To undo the effects of join(), call the function split():

```
$y = "words::separated::by::colons";
@x = split(/::/, $y);
```

13. dec2bin: Converting a Decimal Number to Binary

Here we summarize our understanding of array handling functions with an illustrative script that converts a input decimal number into its binary equivalent. The script logic is to repeatedly divide the number by two and collecting the remainders and finally printing the reverse of all the collected remainders. The script is as follows:

```
#!/usr/bin/perl
foreach $num (@ARGV) {
  $temp = $num;
  until ($num == 0) {
```

```
    $bit = $num % 2;
    unshift(@bit_arr, $bit);
    $num = int($num/2);
}
  $binary_num = join(“”,@bit_arr);
  print (“Binary form of $temp is $binary_num\n”);
  splice(@bit_arr, 0, $#bit_arr+1);
}
```

The output of the above script (assuming script name is dec2bin.pl) is,

```
$ dec2bin.pl 10
Binary form of 10 is 1010

$ dec2bin.pl 8 12 15 10
Binary form of 8 is 1000
Binary form of 12 is 1100
Binary form of 15 is 1111
Binary form of 10 is 1010

$
```

14. grep: Searching an array for pattern

grep function of perl searches an array for a pattern and returns an array which stores the array elements found in the other array.
Example:
$found_arr = grep(/^$code/, @dept_arr); # will search for the specified $code at the beginning of the element in the array @dept_arr.

15. Associative Arrays

In ordinary arrays, you access an array element by specifying an integer as the index:
@fruits = (9, 23, 11);
$count = $fruits[0];   # $count is now 9
In associative arrays, you do not have to use numbers such as 0, 1, and 2 to access array elements. When you define an associative array, you specify the scalar values you want to use to access the elements of the array. For example, here is a definition of a simple associative array:
%fruits=("apple", 9, "banana", 23, "cherry", 11);
It alternates the array subscripts and values in a comma separated strings. i.e., it is basically a key-value pair, where you can refer to a value by specifying the key.
$fruits{“apple”} will retrieve 9. $fruits{“banana”} will retrieve 23 and so on.
Note the use of {} instead of [] here.

There are two associative array functions, keys and values.
keys: Holds the list of subscripts in a separate array.
values: Holds the value of each element in another array.

Normally, keys returns the key strings in a random sequence. To order the list alphabetically, use sort function with keys.

1. foreach $key (sort(keys %region)) { # sorts on keys in the associative array, region
2. @key_list = reverse sort keys %region; # reverse sorts on keys in assoc. array, region

16. Regular Expressions and Substitution

perl supports different forms of regular expressions we have studied so far. It makes use of the functions s and tr to perform substitution and translation respectively.

The s function: Substitution

You can use the =~ operator to substitute one string for another:

```
$val =~ s/abc/def/;     # replace abc with def
$val =~ s/a+/xyz/;      # replace a, aa, aaa, etc., with xyz
$val =~ s/a/b/g;        # replace all a's with b's;It also uses the g flag for global
                        # substitution
```

Here, the s prefix indicates that the pattern between the first / and the second is to be replaced by the string between the second / and the third.

The tr function: Translation

You can also translate characters using the tr prefix:

```
$val =~ tr/a-z/A-Z/;      # translate lower case to upper
```

Here, any character matched by the first pattern is replaced by the corresponding character in the second pattern.

Using Special Characters in Patterns

The following examples demonstrate the use of special characters in a pattern.

1. The * character matches zero or more of the character it follows:
   /jk*l/ # This matches jl, jkl, jkkl, jkkkl, and so on.
2. The + character matches one or more of the preceding character:
   /jk+l/ # This matches jkl, jkkl, jkkkl, and so on.
3. The ? character matches zero or one copies of the preceding character:
   /jk?l/ # This matches jl or jkl.
4. If a set of characters is enclosed in square brackets, any character in the set is an acceptable match:
   /j[kK]l/ # matches jkl or jKl
5. Consecutive alphanumeric characters in the set can be represented by a dash (-):

/j[k1-3K]l/ # matches jkl, j1l, j2l, j3l or jKl

6. You can specify that a match must be at the start or end of a line by using ^ or $:
   /^jkl/ # matches jkl at start of line
   /jkl$/ # matches jkl at end of line
7. Some sets are so common that special characters exist to represent them:
   \d matches any digit, and is equivalent to [0-9].
   \D doesn't match a digit, same as [^0-9].
   \w matches any character that can appear in a variable name; it is equivalent to
   [A-Za-z0-9_].
   \W doesn't match a word character, same as [^a-zA-Z0-9_]
   \s matches any whitespace (any character not visible on the screen); it is equivalent to [\r\t\n\f].

perl accepts the IRE and TRE used by grep and sed, except that the curly braces and parenthesis are not escaped.

For example, to locate lines longer than 512 characters using IRE:
perl –ne 'print if /.{513,}/' filename # Note that we didn't escape the curly braces

Editing files in-Place

perl allows you to edit and rewrite the input file itself. Unlike sed, you don't have to redirect output to a temporary file and then rename it back to the original file.

To edit multiple files in-place, use –I option.

```
perl –p –I –e "s/<B>/<STRONG>/g" *.html *.htm
```

The above statement changes all instances of <B> in all HTML files to <STRONG>. The files themselves are rewritten with the new output. If in-place editing seems a risky thing to do, oyu can back the files up before undertaking the operation:

```
perl –p –I .bak –e "tr/a-z/A-Z" foo[1-4]
```

This first backs up foo1 to foo1.bak, foo2 to foo2.bak and so on, before converting all lowercase letters in each file to uppercase.

17. File Handling

To access a file on your UNIX file system from within your Perl program, you must perform the following steps:
1. First, your program must open the file. This tells the system that your Perl program wants to access the file.
2. Then, the program can either read from or write to the file, depending on how you have opened the file.
3. Finally, the program can close the file. This tells the system that your program no longer needs access to the file.

To open a file we use the open() function.

```
open(INFILE, "/home/srm/input.dat");
```

INFILE is the file handle. The second argument is the pathname. If only the

filename is supplied, the file is assumed to be in the current working directory.

```
open(OUTFILE,">report.dat");     # Opens the file in write mode
open(OUTFILE,">>report.dat");   # Opens the file in append mode
```

The following script demonstrates file handling in perl. This script copies the first three lines of one file into another.

```
#!/usr/bin/perl
open(INFILE, "desig.dat") || die("Cannot open file");
open(OUTFILE, ">desig_out.dat");
while(<INFILE>) {
  print OUTFILE if(1..3);
}
close(INFILE);
close(OUTFILE);
```

18. File Tests

perl has an elaborate system of file tests that overshadows the capabilities of Bourne shell and even find command that we have already seen. You can perform tests on filenames to see whether the file is a directory file or an ordinary file, whether the file is readable, executable or writable, and so on. Some of the file tests are listed next, along with a description of what they do.

if -d *filename*	True if file is a directory
if -e *filename*	True *if* this file exists
if -f *filename*	True if it is a file
if -l *filename*	True if file is a symbolic link
if -s *filename*	True if it is a non-empty file
if -w *filename*	True if file writeable by the person running the program
if -x *filename*	True if this file executable by the person running the program
if -z *filename*	True if this file is empty
if -B *filename*	True if this is a binary file
if -T *filename*	True if this is a text file

19. Subroutines

The use of subroutines results in a modular program. We already know the advantages of modular approach. (They are code reuse, ease of debugging and better readability).

Frequently used segments of code can be stored in separate sections, known as subroutines. The general form of defining a subroutine in perl is:

```
sub procedure_name  {
        # Body of the subroutine
}
```

Example: The following is a routine to read a line of input from a file and break it into words.

```
sub get_words {
```

```
        $inputline = <>;
        @words = split(/\s+/, $inputline);
}
```

Note: The subroutine name must start with a letter, and can then consist of any number of letters, digits, and underscores. The name must not be a keyword.

Precede the name of the subroutine with & to tell perl to call the subroutine.
The following example uses the previous subroutine get_words to count the number of occurrences of the word “the”.

```
#!/usr/bin/perl
$thecount = 0;
&get_words;             Call the subroutine
while ($words[0] ne "") {
  for ($index = 0; $words[$index] ne "";
$index += 1) {
   $thecount += 1 if $words[$index] eq "the";
}
&get_words;
}
```

Return Values

In perl subroutines, the last value seen by the subroutine becomes the subroutine's return value. That is the reason why we could refer to the array variable @words in the calling routine.

Conclusion

Perl is a programming language that allows you to write programs that manipulate files, strings, integers, and arrays quickly and easily. perl is a superset of grep, tr, sed, awk and the shell. perl also has functions for inter- process communication. perl helps in developing minimal code for performing complex tasks. The UNIX spirit lives in perl. perl is popularly used as a CGI scripting language on the web.